32-bit general purpose microcontroller core
User manual

Summary

The document describes Osinka32 MCU: the internal architecture including registers, set of commands and different working modes. It shows how MCU can be integrated into the system with different periphery modules and communication between them. All explanations are given both from software and hardware sides.

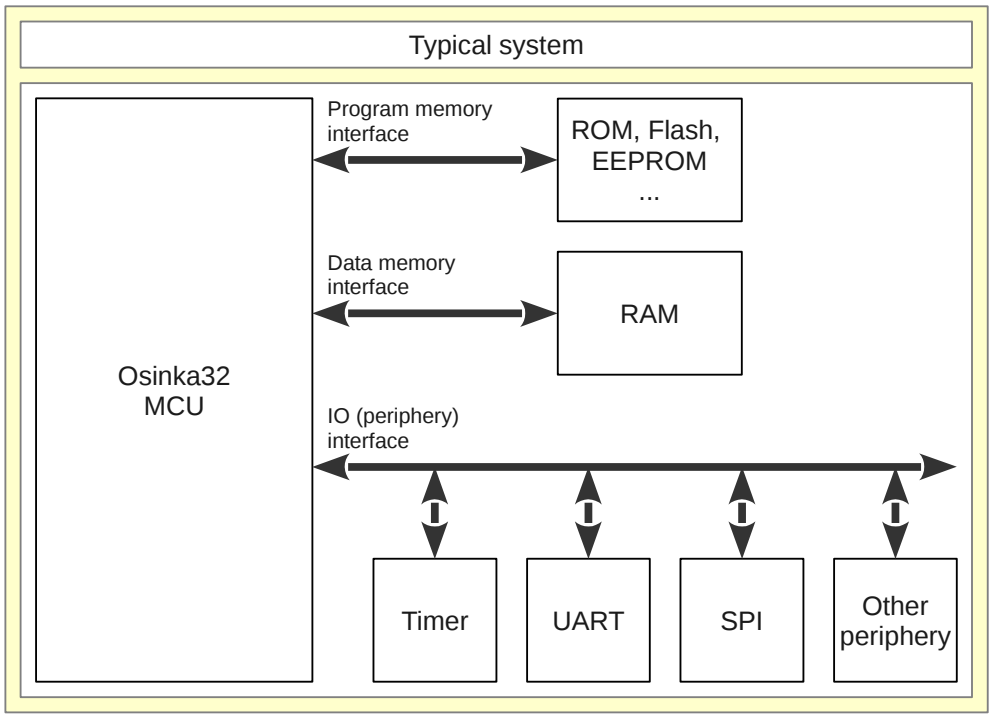
There will also be a description of development tools, assembler directives and step-by step instructions of how to build a project.

Table of Contents

1 Introduction....4
2 Internal structure....5
3 Programming osinka32....6
3.1 Register description....6
3.2 Structure of the assembler file....8
3.3 Memory addressing....9
3.3.1 Compiler directives and variable declaration....9
3.3.2 Address calculation and assembler mnemonics....11
3.3.3 Sequential memory reading and writing....11
3.3.4 Reading data from code memory....12
3.3.5 Memory alignment....12
3.3.6 Data memory organization....13
3.3.7 Code memory organization and execution start point....13
3.4 IO bus and communication with periphery units....17
3.5 Thread switch....17
3.5.1 Purpose and idea....17
3.5.2 Thread context and thread list....18
3.6 Interrupts (IRQ)....21
4 Command summary....22
5 IDE description....25
5.1 System requirements....25
5.2 Installation....26
5.3 Deinstallation....26
5.4 IDE main window....26
5.5 Starting a new project....26
5.6 Compiling the project....28
6 Output files format....30
7 Hardware implementation....31
7.1 External interface....31
7.2 Clock and Reset....32
7.3 Memory interface....33
7.4 IO Interface....35

1 Introduction

Osinka32 is a high performance, easy to use microcontroller core targeted for working as an embedded processor in different applications. It has a rich variety of commands and is able to execute up to 2 commands per clock. MCU operates with 8, 16 and 32 bit data. Executable program and data are located in different memory spaces. Maximal program size is 8Gbytes, maximal data memory addressing range is 4 Gbytes. MCU has an additional 64K IO space which allows connecting different periphery modules.

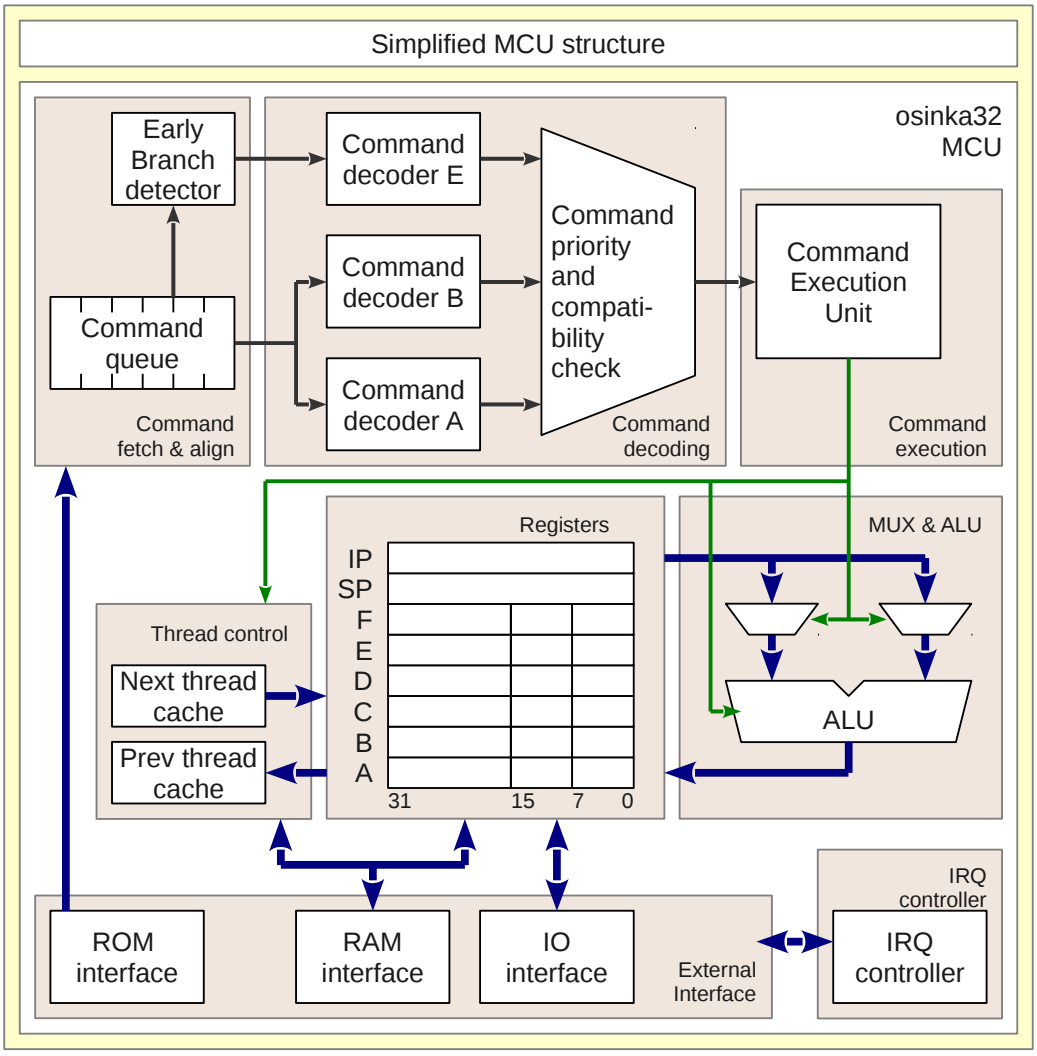


Osinka32 has a hardware multithreading support.

IRQ handling is fast and in general the switch from thread routine to IRQ routine does not require any clock cycle, meaning that if last thread command is executed in clock N, the first IRQ routine command can be executed in clock N+1. IRQ routine does not need to save registers because they are saved automatically by the hardware when interrupt is handled. The register content is restored upon exit from IRQ routine. Saving and restoring of register content does not require any clock cycle and is done by the hardware in the background.

In the document the code memory space will be called ROM and the data memory space will be called RAM.

2 Internal structure



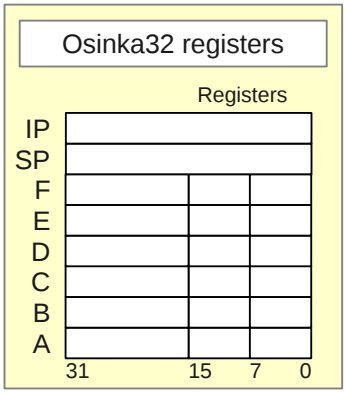
MCU consists of several blocks:

- External interface is used to communicate with RAM, ROM and IO
- Interrupt controller receives and prioritizes interrupt requests
- Command fetch and align controller constantly fills command queue with commands loaded from ROM, detects potential JMPs in the queue and tries to pre-decode them
- Command decoder takes commands from queue, decodes them and checks command compatibility trying to execute 2 commands per clock
- Command execution unit provides execution of decoded commands
- MUX and ALU block selects registers (operands) and executes arithmetical or logical operations
- Registers contain data (operands), pointers and command execution results
- Thread controller provides switch from one thread to another for multi-thread support, loads and stores thread context in the background (when RAM bus is not free)

3 Programming osinka32

3.1 Register description

MCU has 8 32-bit registers. Each of them can be split and a part (8 or 16 bit) of register can be addressed.



Each row is named by letters: A, B, C, D, E, F, ESP and EIP. Each column is addressed by a suffix.

Naming of registers		
Column (bits)	Suffix	Example
7:0	L	EL is the lowest part of E register (bits 7:0)
15:8	H	DH addresses bits 15:8 o D register
15:0	X	CX is lower 16-bit part of C register (bits 15:0)
31:16	W	BW is higher 16-bit part of B register (bits 31:16)
31:0	WX	AWX is a complete 32-bit A register (bits 31:0)

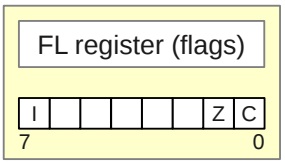
Some registers have special function.
Register FL contains flags. Some of flags represent the result of the last operation, another is used to enable and disable interrupt request.
Register IP is Instruction pointer. It contains the address of a command which will be executed at the next clock. This register is changed (incremented by the command size) automatically during execution.

Remark

It is not likely that SW will use 8 or 16-bit parts of registers IP and SP

Example

'MOV' command can be used to load value to the register. For example '**mov al, 0x08**' loads 0x08 to AL register.
'**mov ah, 0x01**' loads 0x01 to AH register. After execution of these 2 commands AX register will contain 0x0108. The same effect we will have if we execute '**mov ax, 0x0108**'.
Command '**mov bw, ax**' will load the current value of AX into BW register. In this example after the execution of this command BW will contain 0x0108.
'**mov cx, ah**' cannot be executed, because the length of the operands is not the same. Compiler will give an error report.



Flag register contain 3 special bits:
I (bit 7) – enables IRQ requests when set
Z (bit 1) – shows that the result of the last operation is Zero
C (bit 0) – shows that operation caused carry, contains a shift or rotation result

All other bits can be used by the application

Handling and values of the flags will be explained later in this document with the description of commands. Some commands modify flags, some commands don't.

Example

SUB command can be used to subtract constant from the value loaded into the register. Let's assume that BL register contains a value of 5.
'**sub bl, 5**' will set flag 'Z' indicating that the result of operation is Zero.

MOV command does not modify flags.
'**mov bl, 3**' will not change any flag, FL register will keep its previous value.

When FL register is used as a source, operations on it are the same as with any other registers.

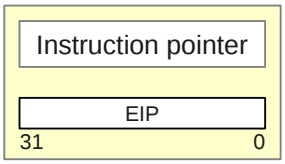
Example
'**mov al, fl**' will copy the value of FL register into AL.
This can be useful to access memory depending on a result of previous operation, where flags Z and/or C will be used to compose an address to access. See memory access commands for more details.

Following piece of code illustrates how to add flag 'C' to AH register

```
mov al, fl
and al, 0x01
add ah, al
```

When FL register is used as a destination, the values of the flags Z and C are logical OR of the previous value of the flag and the result of the operation. Though this is possible, in the SW using C and Z bits of F register as a destination should be avoided because it is difficult to imagine the exact result of these bits. The exception is MOV command, when result values can be easily predicted.

Example
'**mov fl, 0x80**' will enable interrupts.



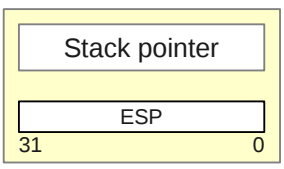
EIP register is the address of the instruction which will be executed in the next clock. This register is incremented automatically by the length of the command with parameters each time MCU executes a command.

Though 8- and 16-bit parts of instruction pointer register can also be addressed (for example, we can address IPL as a lowest 8-bit part of the register) it is not recommended to do so.

EIP register can be freely used in the SW as a source. Any use of this register as a destination will instruct MCU to clean the instruction queue and to continue execution from the address loaded into EIP register.

Example
'**mov eip, 0x1234ABCD**' will instruct MCU to clean the command queue and continue execution from the command located in address 0x1234ABCD. '**mov eip, <value>**' is the equivalent of '**JMP**' instruction. In fact, '**jmp <label>**' is just another, more readable mnemonic of '**mov eip, <value>**' command.

User should avoid using direct loading into the EIP register but use versions of the JMP and CALL instructions instead (which will be described later in the document). Of course, sometimes it is not possible or bigger flexibility is needed. For example, SW designers can keep the procedure address table in the memory and execute JMP or CALL using pointers in this table. This method is fast, very flexible and well readable and can be considered as a good style of programming.



ESP register is the General Purpose register. It can be freely used by the SW in any operations as a source and as a destination. In the IDE there are commands 'push' and 'pop' which use ESP register as a stack pointer.

8- and 16-bit parts of stack pointer register are addressable. If application does not use stack it can freely use ESP as a general purpose register, even parts of it. For example, SPH addresses bits 15:8 of ESP.

Remark

'**PUSH**' and '**POP**' commands are the pseudonyms of more common memory addressing commands:
'push <register>' = 'mov [--esp], <register>'
'pop <register>' = 'mov <register>, [esp++]'

User can implement the equivalent of them using any other register

3.2 Structure of the assembler file

The description of the assembler file structure will be given here by the example. Later in this document all elements will be explained in details.

Example of assembler file structure

Element	Source text
1	`; *** Main.asm ***`
2	`Public Start  ; Label 'Start' is declared here but should be visible from outside`
3	`Extern IrqTimer  ; Label 'IrqTimer' is declared in another module, but will be used here`
4	`#include “IoDef.h”`
5	`.data  ; Data segment`
6	`.org 16  ; Keep 16 bytes free at the beginning`
7	`FFlags:  ; Variable declaration`
8	`db 0`
7	`FDataBuffer:`
8	`db 0 dup(64)  ; 64-bytes buffer`
9	`Align 4`
8	`dd 0 dup(16)  ; 32-bytes stack`
7	`FMainStack:`
5	`.code  ; Code segment`
6	`.org 0xF000  ; Start execution from address 0xF000 (* see comment below)`
10	`Start:`
12	`sjmp Main`
12	`Align 4`
13	`dd IrqTimer  ; IrqVector 1`
13	`dd IrqUart  ; IrqVector 2`
10	`Main:`
11	`mov esp,FMainStack`
11	`mov al,0x80`
11	`mov [FFlags],al`
	`....`
	`CAlphaConst:`
	`db 'Alpha', 0`

Explanation of elements.

1. Comment. Starts with ';'. Can be placed at the end of the line.
2. Public declaration. Label 'Start' is implemented in this module, but must be accessed from another module.
3. External declaration. Label 'IrqTimer' is declared and implemented in another module of the project but it is used here in this module.
4. Include files usually contain definitions common for entire project. It can be different constants, definitions of IO ports, etc.
5. **.data** is a declaration of the data segment. It is a place in the memory where internal variables, buffers, stack are declared. **.code** is a declaration of code segment. Usually it contains executable

code and constants.

6. **.org** directive placed immediately after the declaration of the segment (**.code** or **.data**) specifies the physical location of the segment in memory. For this example, data will be located starting with address 16 (0x0010) and code segment will be started from address 0xF000.
7. Label declaration. Any name with a semicolon at the end declares a label. Name cannot contain spaces, cannot start from a number and must contain only alphanumeric symbols. Labels are used to declare variables, buffers, procedure names, JMP targets etc.
8. Space reservation. **db** reserves 1 byte; **dw** reserves 2 bytes (1 word); **dd** reserves 4 bytes. If directive **dup** is placed after space reservation it shows how many bytes, words or double words to reserve. In the example above 'FFlags' occupies only 1 byte; 'FDataBuffer' occupies 64 bytes; 'FMainStack' occupies 64 bytes (4*16=64); 'CAlphaConst' occupies 6 bytes. Initialization of the variables and arrays in the data segment often has no effect: if we write '**FFlags: db 4**' it doesn't always mean that at the beginning of the execution 'FFlags' variable will contain 4. It depends whether there is a mechanism of initialization of RAM location or not in your system. Initialization of constants in code segment is always done (otherwise program will not work). So, it is guaranteed that 'CAlphaConst' will be initialized with a zero-terminated string 'Alpha'. Note that label 'FMainStack' is placed after the space reservation and not before. This will be explained later.
9. Alignment of the element. If we write **Align 2** it forces a compiler to place following element at the even address. If we write **Align 4**, the address of the following element will be divisible by 4: 0x0000, 0x0004, 0x0008 and so on. See 'Memory alignment' chapter for details.
10. Declaration of label. Similar to the data declaration, label 'Start' declares a start of the procedure or any executable code. It can also be a target for 'JMP' command.
11. Executable code commands. Each command occupies 1 line. There can be a comment at the end of the line (started with ';' of course). The full command list will be given later in this document.
12. If interrupts are used the interrupt address table must be declared. In this case immediately after the executable entry point label ('Start:' label in this example) place short jump command to the first command to be executed and a line 'Align 4'. The first command executed after reset will be 'sjmp Main'.
13. Interrupt vector table. This table must be aligned by 4 and contain addresses of IRQ routines corresponding to the interrupt source. Note that IRQ[0] does not exist in osinka32 and the first interrupt has index 1 (and not 0). In this example there are 2 interrupts: IrqTimer and IrqUart.

** org directive for data and code segments is given only as an example. In most cases both data and code segments start with address 0 (examples you may download from the site). So directive org is not needed.*

3.3 Memory addressing

3.3.1 Compiler directives and variable declaration

Implementation of microcontroller systems usually requires 2 types of memory:

- Memory, which contains executable code and constants (code memory);
- Memory, which contains data, variables, buffers, stack, etc (data memory).

Code memory is implemented as ROM, EEPROM, Flash. It can be implemented as RAM in FPGAs, but has to be loaded with executable code. Data memory is implemented as RAM.

In order to start code segment directive **.code** is used. Directive **.data** starts data segment. In the same source file there can be several pieces of code and several pieces of data. If memory segment is not specified, an error message will be given by the compiler. During compilation all pieces of data and code will be grouped together and one linear binary file will be produced. See binary description for details.

Compiler directives	
Directive	Description
.code	Starts code segment
.data	Starts data segment
db	Reserves 1 byte for the variable
dw	Reserves 2 bytes for the variable
dd	Reserves 4 bytes for the variable
dup	Specifies the length of the array
Align	Aligns following data
.org	Specifies the physical address of mapping
Public	Makes a label, declared in this module, visible from other modules
Extern	In order to use label declared in another module it must be declared by 'Extern' directive

'Memory mapping' chapter will explain how to map code and data to exact physical location.

Example of the assembler file (Similar to the example in 'Structure of the assembler file' chapter)

```
Public Start          ; Label 'Start' is declared here but should be visible from outside
Extern IrqTimer       ; Label 'IrqTimer' is declared in another module, but will be used here

#include “IoDef.h”

.data                 ; Data segment
.org 16               ; Keep 16 bytes free at the beginning

FFlags:               ; Variable declaration
    db  0

FDataBuffer:
    db  0 dup(64)     ; 64-bytes buffer

  Align 4
    dd  0 dup(16)     ; 32-bytes stack
FMainStack:

.code                 ; Code segment
.org 0xF000           ; Start execution from address 0xF000 (* see comment below)

Start:
        sjmp    Main
  Align 4
    dd  IrqTimer        ; IrqVector 1
    dd  IrqUart         ; IrqVector 2

Main:
        mov     esp,FMainStack
        mov     al,0x80
        mov     [FFlags],al
        ....
```

Note that since stack grows down, label 'FMainStack' is declared after the reservation of the space. All other labels are declared before the space reservation.

In the example above we loaded a pointer to FMainStack into ESP register using instruction
“**mov esp,FMainStack**”.
We initialized a value of FFlags using instruction
“**mov [FFlags],al**”.
Note that we use square brackets when we need to access a value of the variable and we don't use brackets to get a pointer to the variable. This will be explained in the following chapter.

3.3.2 Address calculation and assembler mnemonics

In the assembler we will use square brackets '[' and ']' in order to say that value specifies the address in memory.

Example

Let's assume that register BWX=0x00001234. At the address 0x00001234 in the memory we had previously written 0xFF10AE54.
'mov awx, bwx' will load the value of BWX into AWX register. AWX will contain 0x00001234.
'mov awx, [bwx]' will read 4 bytes at the memory location 0x00001234 and load this value into AWX. AWX will contain 0xFF10AE54.

Illustration for the example

Command	Memory (same for both commands)	Before execution	After execution	Comment
`mov awx,bwx`	Address -> Data 0x1234 -> 0x54 0x1235 -> 0xAE 0x1236 -> 0x10 0x1237 -> 0xFF 0x1238 -> 0x55 . . .	awx=0x00000000 bwx=0x00001234	awx=0x00001234 bwx=0x00001234	Value of BWX is copied into AWX register
`mov awx,[bwx]`		awx=0x00000000 bwx=0x00001234	awx=0xFF10AE54 bwx=0x00001234	Value of memory at the address in BWX register is copied into AWX register

There are several possibility to specify address. Any of 32- or 16-bit register (except of IP) can be used by the SW to hold a pointer. We can also use a pair of registers (base+index), a register and a constant. Following table gives all possible combinations. '#' indicates a constant. 32-bit constant is used in a pair with a 32-bit register; 16-bit constant makes a pair with a 16-bit register. Note that if 16-bit register is used, the addressable range cannot exceed 64K.

Possible combinations to compose an address

Register	Example	Comments
a	`mov b,[ax]`	
b	`mov ax,[bwx]`	
c	`mov ewx,[cwx]`	
d	`mov al,[dx]`	
e	`mov [ewx],al`	
f	`mov [fx],fw`	
sp+#	`mov ax,[esp+2]`	Since stack grows down, constant in this command is >0 ('mov ax, [sp - 2]' does not exist)
#	`mov ew,[FData]`	FData is the variable declared somewhere in the application
a+#	`mov [FDataBuf+awx],bh`	FDataBuf is the name of an array declared somewhere in the application
b+#	`mov [bx+2],bx`	
c+#	`mov ax,[cwx-4]`	
d+#	`mov al,[dx+3]`	
b+c	`mov ax,[bwx+cwx]`	With of registers which form a pair must be identical, i.e. Both registers must be of 32-bit or both registers must be of 16-bit.
b+d	`mov al,[bx+dx]`	
e+c	`mov [ewx+cwx],ah`	
e+d	`mov al,[ex+dx]`	

3.3.3 Sequential memory reading and writing

When dialing with arrays it is often necessary to access their elements sequentially. There is a set of commands which will automatically increment or decrement the pointer. When command access 2-bytes

value, pointer will be incremented or decremented by 2. If 4 bytes is accessed, the pointer will be incremented or decremented by 4. Only post-incrementing and pre-decrementing are allowed.

In the assembler we will place '++' after the pointer in order to show that address has to be incremented after the operation. We will place '--' before the pointer to force the address to be decremented before operation.

Examples of sequential memory access commands			
Example	Before execution	After execution	Comment

`mov al,[bx++]`	`al = 0x00`
	`bx = 0x1000`
	`Memory:`
	`0x1000 -> 0x12`
	`0x34`
	`0x10`
		`al = 0x12`
		`bx = 0x1001`
		`Memory:`
		`0x1000 -> 0x12`
		`0x34`
		`0x10`
			Address (in this case it is a BX register) is incremented after execution

`mov [cwx++],ax`	`ax = 0x8055`
	`cwx = 0x1000`
	`Memory:`
	`0x1000 -> 0x12`
	`0x34`
	`0x10`
		`ax = 0x8055`
		`cwx = 0x1002`
		`Memory:`
		`0x1000 -> 0x55`
		`0x80`
		`0x10`
			After execution of this command address (CWX register) is incremented by 2, because we have written 2 bytes

`mov ew,[--dx]`	`ew = 0x8055`
	`dx = 0x1034`
	`Memory:`
	`0xAE`
	`0x24`
	`0x1034 -> 0x14`
		`ew = 0x24AE`
		`dx = 0x1032`
		`Memory:`
		`0xAE`
		`0x24`
		`0x1034 -> 0x14`
			Address (DX register) is decremented by 2 before execution of the command and data will be read from the new address: 0x1032

`mov [--ewx],dwx`	`ed = 0x8055` `iy = 0x1033`	This command will cause an error, because address of 4-byte element is not aligned	We are trying to write 4-byte data to non-aligned address. MCU will not perform this operation. Take special care to align all variables, arrays, etc.

3.3.4 Reading data from code memory

Application can keep some data in the code memory (ROM, EEPROM, Flash). It is very useful to store there some initialization data, constants, etc. In order to read code memory special command **ldc** is used. Only b, c, d and e registers (32- or 16-bit) can be used as an address.

```
.code                   ; Code segment

CConstString:           ; Some constant
    db  'Test string',0

  Align 2               ; do not forget to align executable code

ProcTest:
        . . .
        mov     bwx,CConstString  ; Initialize pointer (BWX in this case)
        ldc     al,[bwx]          ; After this operation al will contain 'T'
```

3.3.5 Memory alignment

MCU cannot read or write 2-byte variables if their address is not aligned by 2 and 4-byte variables if their address is not aligned by 4. This rule applies to all data both in data segment and in code segment. This rule applies for all arrays of 2-byte and 4-byte elements. Since push and pop commands which access stack work with 4-byte data, stack must be aligned by 4.

In case of incorrect alignment compiler will give a warning, but this warning is sometimes difficult to interpret. See following example.

```
.data                    ; Data segment
  StartAddr 16           ; 16 bytes is occupied by interrupt vectors

FFlags:
    db  0

FDataWord:               ; not aligned!
    dw  0

.code                    ; Code segment

ProcTest:
        mov     ax,0x1234
        mov     [FDataWord],ax  ; will cause an error
```

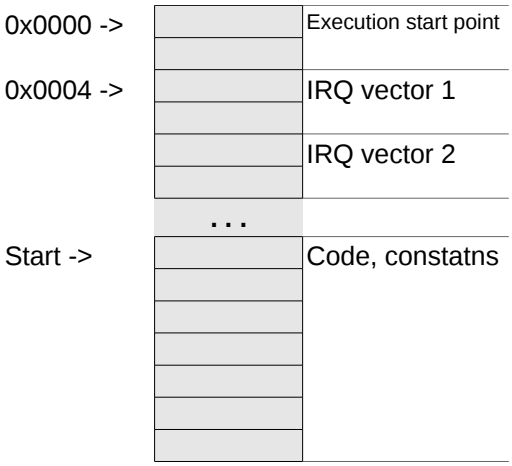
3.3.6 Data memory organization

Data memory is organized as a linear array. Start address depends on digital organization but usually starts from 0. Maximal addressable range is 4Gbytes.

3.3.7 Code memory organization and execution start point

Code memory is organized as a linear array. Start address depends on digital settings. It is 0 by default. In order to change this value, change the parameter CstartAddr in mcdCore.v and recompile the project.

Maximal addressable range is 8Gbytes.



Interrupt vectors are located at the beginning of the memory starting with address 0x0004. Each interrupt vector occupies 4 bytes of address. Maximal number of interrupts supported by the MCU is 7. Interrupt 0 does not exist.

If interrupts are used, place command
sjmp Start
at the beginning of executable code (example will be given).

The order in which files are compiled is important. Execution start point must be located in a file which is linked first.

Assembler directive **.org** is used to specify the physical address in memory where the segment has to be located. This address has to be specified for both **.data** segment and **.code** segment. It is enough to specify this address only once in the project, but it has to be specified in the file which will be compiled first. This will be clarified by following examples.

Example
Let's imagine that the project consists of only one source file. 2 IRQ vectors are used to handle interrupts: IrqTimer (IRQ 1) and IrqUart (IRQ 2).

Both data and code segments are physically located at address 0. So directive

.org can be omitted. See following source code.

```
.data                   ; Data segment
    dw  0 dup(16)
FStack:

FFlags:
    db  0

FFifoBuf:
    db  0 dup(16)

.code                   ; Code segment
        sjmp    Start   ; Execution starts here

  Align 4               ; It is important to align IRQ table by 4
        ; ** IRQ table starts here
     dd IrqTimer        ; Pointer to the interrupt routine IrqTimer
     dd IrqUart         ; Pointer to the interrupt routine IrqUart
   Start:
        mov     esp,FStack   ; Execution of program will start here
        mov     al,0
        mov     [FFlags],al
        . . .

IrqTimer:
        ; Implementation of IRQ routine is here
        iret

IrqUart:
        ; Implementation of IRQ routine is here
        iret
```

Example

Very often at the beginning of the execution it is necessary to initialize an IRQ vector table. Let's start another project which is similar to the previous one but which consists of 2 source files: 'Main.asm' and 'IrqVect.asm'. In the first file we keep an entry point and in the second one we will implement IRQ handlers. There will be 2 IRQ handlers: IrqTimer (mapped on vector 0) and IrqUart (mapped on vector 1).

Source file **Main.asm**

```
Extern IrqTimer
Extern IrqUart

.data                   ; Data segment
    dw  0 dup(16)
FStack:

FFlags:
    db  0

FFifoBuf:
    db  0 dup(16)

.code                   ; Code segment
        sjmp    Start   ; Execution starts here

  Align 4               ; It is important to align IRQ table by 4
        ; ** IRQ table starts here
     dd IrqTimer        ; Pointer to the interrupt routine IrqTimer
     dd IrqUart         ; Pointer to the interrupt routine IrqUart
   Start:
        mov     esp,FStack   ; Execution of program will start here
        mov     al,0
        mov     [FFlags],al
        . . .
```

Source file **IrqVect.asm**

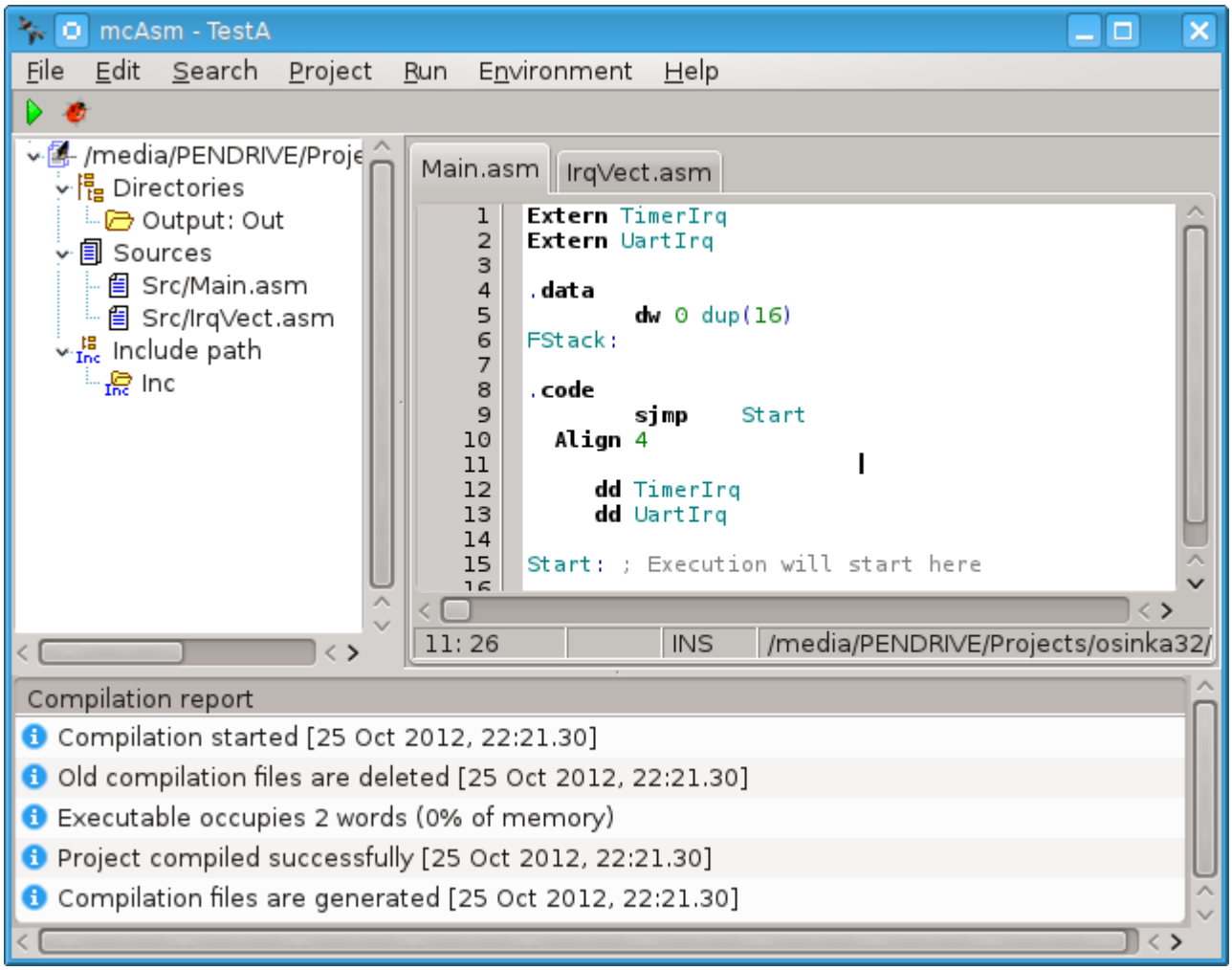
```
Public IrqTimer
Public IrqUart

.code                   ; Code segment

IrqTimer:
        ; Implement code here
        iret

IrqUart:
        ; Implement code here
        iret
```

Remark
In the example above, file 'Main.asm' must be first in the project source list; file 'IrqVect.asm' must be second. If it is not a case, move file up or down in the list. Details of using IDE will be described later. See following screen shot.



^ Screen-shot above is made under Linux Fedora-16

3.4 IO bus and communication with periphery units

IO bus is provided for addressing periphery units such us timers, UART ports, SPI, Codecs, etc. Digital designers can map all these devices to the memory address space as well, but separate IO bus has several advantages.

- Address decoding is easier (less address lines to decode);
- Performance of the system is better (usually the slowest path goes through the memory, additional multiplexers bring bigger delay);
- Command size is smaller.

There are several disadvantages.

- There are lower variety of commands to work with IO than with memory;
- Register choice to work with IO is restricted (only B register can be used as a port address, only A register can hold data).

IO port can be addressed by a 16-bit register or a 8-bit constant. If a 8-bit constant is used, port address cannot exceed 255. Only 4 16-bit registers are available as a port address: AW, BX, BW, DW.

Only A register (AL, AX or AWX) can be used to hold data.

IO access commands		
Command	Example	Comment
`in a,<reg>`	`in al,dw`	Data from port addressed by DW register will be read to AL
`in a,#8`	`in ax,0x13`	Data will be read from port #0x13 and be placed into AX register
`out <reg>,a`	`out bx,awx`	Data, previously placed into AWX register will be written to the port addressed by BX register
`out #8,a`	`out 0x12,awx`	Data from AWX register will be written to the port #0x12

Remark
When working with IO space the alignment is not necessary. For example we can output AWX to address 0x13. The hardware periphery is responsible for the data treatment.

We can use the same address to output data of different width, and periphery can perform different functions. By doing so we can save IO space.

There are free periphery examples given with the MCU. They actively use this feature. See periphery examples for details.

3.5 Thread switch

3.5.1 Purpose and idea

Thread switch is a mechanism which allows to execute several processes at the same MCU at virtually the same time. Of course, since there is only one core MCU will execute one process at a given time. But the switch from one process to another is very fast and in general requires 3 clock cycles.

When MCU executes thread switch it suspends active process (the process which is executed now) and starts executing another process. Later the execution of suspended process resumes.

The process does not 'feel' the thread switch because registers stays unchanged.

Thread switch can be initiated by SW: there is a special command **swt** (Switch Thread) which suspends the execution of a process and instructs MCU to start executing next process.

Remark
In this version of Osinka32 thread switch cannot be initiated by hardware.

Similar to jump commands, **swt** can be conditional and unconditional and has a label name as a parameter. Execution will resume from indicated label.

Example
In the following example there is a piece of code which waits data from UART and returns control to other processes if UART is not ready.

There are some variables and constants in the example. Their declaration will not be shown. We just assume that
- IoUartFlags – name of port to read UART flags
- IoUartData – reading from this port reads UART data buffer
- CByteReceived – predefined constant to check UART reception flags

```
    ; Next byte
mWaitByte:
    in      al,IoUartFlags   ; Get UART flags
    test    al,CByteReceived ; Check if byte is received
    swt_z   mWaitByte        ; Pass control to next process / jump to label mWaitByte

mByteReceived:
    in      al,IoUartData    ; Read data from UART
    mov     [bwx++],al       ; Copy data to the buffer
    inc     cw               ; Increase byte counter
```

It will take maximum 9 CLK to switch from one thread to another, but usually the switch takes as less as 3 CLK. This is because MCU can prepare to the thread switch by filling internal cache. If there are enough commands between 2 target labels (in the example above 'mWaitByte') and the **swt_z** command (in the example above there are 2), so MCU will have enough time to fill cache.

3.5.2 Thread context and thread list

Though thread switch is almost transparent for the user it is important to know what Thread context is. Thread context is an image of all registers at the moment thread switch occurs. Thread contexts of all threads are stored in the memory one after another. When thread switch occurs, MCU saves all registers of the current thread, and loads from memory all registers of the next thread in the list executing threads one after another. When it arrives to the end of the list it simply restarts from the first thread in the list.

Before the first **swt** command can be executed, SW must prepare and initialize the Thread list in the memory. The Thread list must be aligned by 8.

Registers in the Thread context are stored in the following order.

The order of registers in the thread context	
Register	Offset
AWX	0
BWX	4
CWX	8
DWX	12
EWX	16
FWX	20
ESP	24
EIP	28

When initializing Thread list SW has to initialize at least EIP and FWX register of each thread. EIP is a starting point from which thread will start execution. FWX must contain zero at least in IF flag, because interrupts at the beginning of thread should be disabled.

Once thread contexts in the list are initialized, application has to place the address to the list into BWX register and execute **sbl** command. This command has one parameter: number of threads in the list minus 1. Note that the thread context with index zero does not have to be initialized because it represents currently working process.

Example

Let's make a project which contains 3 threads:

- Main thread (supervisor)
- Uart thread used to communicate by UART
- Terminal thread used to output some data on the screen

The entry point of Uart thread will be called UartProcess and the entry point of Terminal thread we will call TerminalProcess. Each thread will be implemented in a separate source file for simplicity. So, project will contain 3 files:

- Main.asm
- Uart.asm
- Term.asm

At the beginning of thread list initialization we will fill all registers with zeros to avoid garbage and basically to initialize FWX (Flags). Then we will specify the entry point for each thread.

Source file **Main.asm**

```
Extern UartProcess
Extern TerminalProcess

.data                   ; Data segment

; *** Variable declaration
    dd  0 dup(16)
FStack:

    Align 8
FThreadList:
FThMain:
    dd  0 dup(8)
FThUart:
    dd  0 dup(8)
FThTerminal:
    dd  0 dup(8)

.code                   ; Code segment
Start:
        mov     esp,FStack   ; Execution of program will start here
        ; Thread list initialization
        ; Fill thread list with zeroes
        mov     awx,0
        mov     cl,24    ; 3*8=24 size of entire list
        mov     bwx,FThreadList
        ; fill list with zeroes
    sThListFill:
        mov     [bwx++],awx
        dec     cl
        jnz     sThListFill
        ; Specify entry points
        ; .. Uart
        mov     bwx,FThUart
        mov     awx,UartProcess
        shr     awx,1           ; Code memory is organized by words, so divide address by 2
        mov     [bwx+28],awx    ; 28 is EIP offset
        ; .. Terminal
        mov     bwx,FThTerminal
        mov     awx,TerminalProcess
        shr     awx
        mov     [bwx+28],awx    ; initialize start point
        ; Enable thread mechanism
        mov     bwx,FThreadList
        sbl     2               ; 3-1=2: number of threads in the list

        ; Now we are free to execute swt
        mov     ax,17
    sSupervisor:
        inc     ax
        swt     sSupervisor
```

Source file **Uart.asm**

```
Public UartProcess

.data
    Align 4
    dd  0 dup(16)
FUartStack:

.code
UartProcess:
        mov     esp,FUartStack
        call    UartInit
    upWait:
        in      al,IoUartFlags ; Read UART flags
        test    al,0x80        ; Let's imagine to verify if the data was received we analyze a flag
        swt_z   upWait         ; If UART not ready, switch task and wait. It is just an example...

    upProcessData:
        ; Implement code here
        . . .
```

Source file **Term.asm**

```
Public TerminalProcess

.data
    Align 4
    dd  0 dup(16)
FTermStack:

.code
TerminalProcess:
        mov     esp,FTermStack
        call    TimerInit
    tpWait:
        in      al,IoTimerFlags ; Let's imagine we are waiting some timer event
        test    al,0x01         ; It is just an example
        jnz     tpTimerOver
        test    al,0x02
        jnz     tpSomethingElse
        swt     tpWait
    tpTimerOver:
        ; Implement code here
        . . .
```

In this example we do not initialize ESP for all threads. Thread procedure is responsible to initialize ESP itself in this case.

When we first call **'swt sSupervisor'** in Main function, control will be given to the next thread. In our case it is 'UartProcess'. It will execute its first command '**mov esp,FUartStack**' and continue execution till **'swt_z upWait'**. At this point control will pass to the next thread, Terminal process. Like UartProcess, it will execute its first command '**mov esp,FTermStack**' and continue till **'swt tpWait'**. Since it is the last thread in the list, control will return back to the main process and command '**inc ax**' will be executed. Note that AX keeps its value despite several tasks were already executed in other threads. When main process executes **'swt sSupervisor'**, control is passed to UartProcess and it executes command '**in al,IoUartFlags**' and so on, one task after another.

3.6 Interrupts (IRQ)

Sometimes microcontroller system has to react on the external event and the reaction should be as soon as possible. IRQ mechanism is used for this purpose. IRQ is an external event and associated with this event small procedure-handler.

IRQ routine should be small and fast and should return control to the system as soon as possible. Command **iret** is used to exit IRQ routine and return control to the interrupted process. IRQ routine is free to use any registers. Due to thread mechanism it does not have to save register values. When IRQ routine is called, registers contain garbage except of IF flag which is set to 0. IRQ routine is not allowed to set this flag, i.e. no nested IRQs are allowed. IRQ routine cannot use **swt** command.

In worst case switch from thread execution to interrupt routine execution requires 5 clocks. But this only happens when IRQ is sensed just after **swt** command is executed. In general switch from thread to interrupt routine requires 0 clock cycles.

Before enabling IRQ SW has to initialize thread mechanism. This is because when IRQ is called, all registers are saved into the current thread context. IRQs can be enabled or disabled by the SW individually for each thread.

See 'Memory organization and execution start point' chapter for more details.

4 Command summary

During command description some definitions will be used.

Definitions used for command description			
Definition	Comment	Example	Explanation for the example
`r`	8, 16 or 32 bit register	inc r	Command inc increments register. Any register can be used: 8, 16 or 32 bit
`r8`	Only 8-bit register		
`r16`	Only 16-bit register		
`r32`	Only 32-bit register	pop r32	Command applies only for 32-bit registers: 'pop al' will cause an error
`ra`	Address register	mov ah,[ra]	'ra' is a 16 or 32-bit register which can be used for addressing
`ri`	Port address	Out dw,awx	Only AW, BX, CW, DW registers can be used to address an IO port
`#8`	Only 8-bit constant	mov al,#8	Only 8-bit constant can be used. 'mov al,260' is invalid
`#16`	Only 16-bit constant	mov [bx+#16],ex	Even if we write 'mov [bx+4],ex', constant 4 will occupy 2 bytes
`#32`	Only 32-bit constant	mov [bwx+#32],el	Even if we write 'mov [bwx+5],ex', constant 5 will occupy 4 bytes
`mmm`	Memory location	mov al,[mmm]	mmm specifies a memory location. See following table for more details
`port`	IO port	in al,port(dw)	DW register contains an address of IO port
`nil`	Do not save result	nil=r-#	Subtract constant from register but don't save result (changes only flags)
`&`	Logical AND	dr=dr & sr	
`|`	Logical OR	dr=dr | sr	
`^`	Logical XOR	dr=dr ^ sr	
`NextCmd`	Next command	IP=NextCmd	See flow-control command description. Do nothing, just go to next command
`ThThis`	Current register context	ThThis=ThPrev	Uses to indicate a Thread Context. Used during Thread flow control command description
`ThPrev`	Context of previous thread	ThPrev=ThThis	
`ThNext`	Context of next thread	ThThis=ThNext	

Sometimes instead of 'r' we will use definition 'sr' or 'dr'. 'sr' – to specify the source register, 'dr' – to specify destination. Both source and destination must have same length: both 8-bit, both 16-bit or both 32-bit registers.

Example
Command
mov dr, sr
performs operation **mov** where **dr** is a destination register (8, 16 or 32 bit) and **sr** is a source register (same length as dr).

Possible values of mmm			
16-bit addressing		32-bit addressing	
ax	ax+#16	awx	awx+#32
bx	bx+#16	bwx	bwx+#32
cx	cx+#16	cwx	cwx+#32
dx	dx+#16	dwx	dwx+#32
ex	bx+cx	ewx	bwx+cwx
fx	bx+dx	fwx	bwx+dwx
sp+#16	ex+cx	esp+#32	ewx+cwx
#16	ex+dx	#32	ewx+dwx

Memory can be addressed by different ways: register, constant, a pair of registers and register+constant.

These combinations are listed at the table in the left.

16 or 32-bit addresses can be used. 16-bit address can access only lower 64K of memory.

Many commands have several variations. All of them will be listed. Some commands change flags, others don't. If Flag register is changed by the instruction, it will be indicated.

Data transfer instructions							
Command	Variations	Result	Example	ZF	CF	CLK	Comments
mov	mov dr,sr	dr=sr	mov ax,cx			1	Flags are not changed by this command unless F is a destination register Avoid commands where IP is used as a destination. Use flow control commands instead XX = 1, 2 or 4 depending of the size of a source or destination register: mov al,[ewx++] will increment ewx by 1 mov ax,[ewx++] will increment ewx by 2 Take care of alignment when source or destination is located in memory
	mov dr,#	dr=#	mov dw,0x08			1	
	mov dr,[mmm]	dr=[mem]	mov cl,[dwx]			1/2	
	mov [mmm],sr	[mmm]=sr	mov [FData],bl			1	
	mov dr,[ra++]	dr=[ra] ra=ra+XX	mov al,[awx++]			1/2	
	mov dr,[--ra]	dr=[ra-XX] ra=ra-XX	mov ch,[--ax]			1/2	
	mov [ra++],sr	[ra]=sr ra=ra+XX	mov [ex++],ew			1	
	mov [--ra],sr	[ra-XX]=sr ra=ra-XX	mov [--ewx],bw			1	
out	out ri,sr	port(ri)=al	out aw,al			1	It is advised to create an include file where all port addresses are defined. This file will be common for the entire project
	out #8,sr	port(#8)=al	out 0x10,al			1	
in	in dr,ri	al=port(ri)	in al,dw			1/2	
	in dr,#8	al=port(#8)	in al,IoComm			1/2	
ldc	ldc a,[ra]	a=code [ra]	ldc al,[bwx]			1/2	Read code memory. Only A register can be used as a destination

Push and pop instructions							
Command	Variations	Result	Example	ZF	CF	CLK	Comments

push	push r32	[esp-4]=r32
		esp=esp-4
			push awx			1	push and pop is just another mnemonic for
							mov [--esp],r32
							mov r32,[esp++]

pop	pop r32	[esp]=r32 esp=esp+4	pop cwx			1/2	

Memory read and port read instructions require 1 clock cycle for execution if the result is not required immediately by the next command. Otherwise they require 2 cycles.

Example

There are 2 examples of code which do exactly the same. The 1st one requires 4 clocks for execution and the 2nd one requires 3.

1st example:

```
     mov        al,[bwx]
     and        al,0x01
     mov        cx,0x0008
```

2nd example:

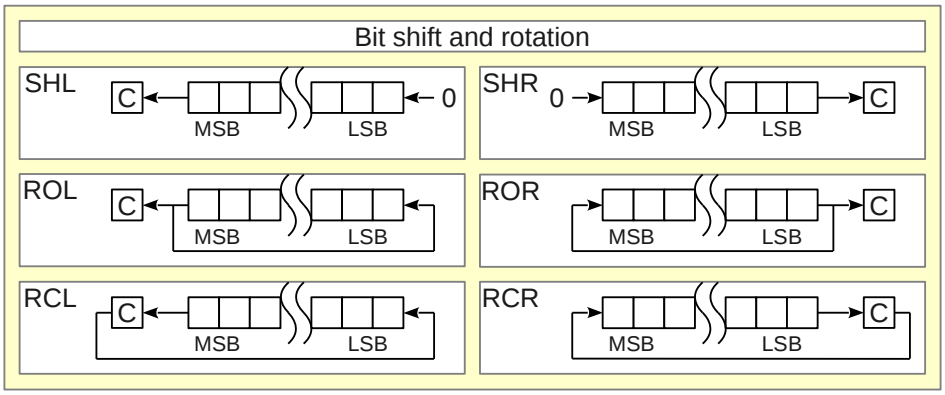
```
     mov        al,[bwx]
     mov        cx,0x0008
     and        al,0x01
```

The difference is in the order of commands. In the 1st example the value read from the memory is required immediately by the next command (and al,0x01) and in the 2nd example the memory read operation is followed by a command where the read value is not used (mov cx,0x0008).

Note that MCU can combine memory read/write operation with other operation and execute 2 commands per clock. In this example this feature is not used.

Arithmetic instructions							
Command	Variations	Result	Example	ZF	CF	CLK	Comments
add	add dr,sr	dr=dr+sr	add ax,cb	+	+	1	When Flag register is a destination of the command, the result for 2 lower bits is different. See 'Register description' chapter for details
	add dr,#	dr=dr+#	add al,4	+	+	1	
sub	sub dr,sr	dr=dr-sr	sub al,e	+	+	1	
	sub dr,#	dr=dr-#	sub c,3	+	+	1	Commands where destination register is IP should be avoided and flow control commands should be used.
cmp	cmp dr,sr	nil=dr-sr	cmp ix,ax	+	+	1	
	cmp dr,#	nil=dr-#	cmp al,0	+	+	1	
and	and dr,sr	dr=dr & sr	and ax,cb	+	+	1	Command **cmp** performs the same operation as **sub**, but doesn't save the result of the operation. This command is used to compare 2 operands
	and dr,#	dr=dr & #	and c,0xFE	+	+	1	
test	test dr,sr	dr=dr & sr	test ax,ax	+	+	1	Command **test** performs the same operation as **and**, but doesn't save the result of the operation. This command is used to check which bits in the register are set
	test dr,#	dr=dr & #	test al,0x80	+	+	1	
or	or dr,sr	dr=dr | sr	or cb,ax	+	+	1	
	or dr,#	dr=dr | #	or al,1	+	+	1	
xor	xor dr,sr	dr=dr ^ sr	xor al,e	+	+	1	
	xor dr,#	dr=dr ^ #	xor e,0x55	+	+	1	

Shift instructions							
Command	Variations	Result	Example	ZF	CF	CLK	Comments
shl	shl r	See explanation below	shl al	+	+	1	
shr	shr r		shr bx	+	+	1	
rol	rol r		rol ew	+	+	1	
ror	ror r		ror dwx	+	+	1	
rcl	rcl r		rcl ah	+	+	1	
rcr	rcr r		rcr fh	+	+	1	



Flow control							
Command	Type	Condition		Result	Max dst	CLK	Comments
jmp #32	Absolute	Always		EIP=#32	Everywhere	2..4	It is difficult to estimate how many clock cycles flow control commands will require for execution. MCU makes all possible efforts to use as less clocks as possible. There is a powerful early branch detection algorithm. Also MCU often executes branch command with another command at the same clock (2 commands per clock). In general all JMP commands require not more than 1 clock cycle both if condition is true and if condition is false. In a very rare case if branch is not detected early and branch command cannot form a pair with another command and a destination command is badly aligned, maximum 4 clock cycles are used.

sjmp #8	Relative	Always		if Condition then
				begin
				EIP=EIP +/- #8
				end
				else
				begin
				EIP = NextCmd
				end
					-512 .. +511 from current IP value	1..4	

jbe #8		CF=1 or ZF=1	<=				
jc #8		CF=1	<				
jnc #8		CF=0					
jz #8		ZF=1	==				
jnz #8		ZF=0	<>				
ja #8		CF=0 and ZF=0	>				
jae #8		CF=0	>=				

call #32	Absolute	Always		if Condition then
				begin
				[esp-4]=EIP
				EIP=#32
				esp=esp-4
				end
				else
				begin
				EIP = NextCmd
				end
					Everywhere	2..4 if condition is TRUE
						
						1 if condition is FALSE	

call_be #32		CF=1 or ZF=1	<=				
call_c #32		CF=1	<				
call_nc #32		CF=0					
call_z #32		ZF=1	==				
call_nz #32		ZF=0	<>				
call_a #32		CF=0 and ZF=0	>				
call_ae #32		CF=0	>=				
ret		Always		EIP=[esp]; esp=esp+4	Everywhere	3..4	
nop		Always		IP = NextCmd		1	

Threads, IRQ					
Command or event	Condition		Result	CLK	Comments
External IRQ request	IF=1		ThPrev=ThThis eip=[IrqIndex<<1]	0..5	See 'Thread switch', 'IRQ control' chapters for details
iret	Always		ThThis=ThPrev	3..4	
swt #32	Always		ThPrev={#32,esp,fwx,ewx,dwx,cwx,bwx,awx} ThThis=ThNext	3..8	Maximal number of clocks supposes that request arrived just after swt command when context of thread (ThPrev) is not yet transferred to the memory AND target command is not aligned by 4. Usually this case is rare
swt_be #32	CF=1 or ZF=1	<=			
swt_c #32	CF=1	<			
swt_nc #32	CF=0				
swt_z #32	ZF=1	==			
swt_nz #32	ZF=0	<>			
swt_a #32	CF=0 and ZF=0	>			
swt_ae #32	CF=0	>=			

Other commands			
Command	Purpose	CLK	Comments
sbl #32	Prepares Thread mechanism	1	See 'Thread switch chapter' for details

5 IDE description

5.1 System requirements

System requirements	
Version	Requirements
win	Windows XP, Vista or Windows-7
lin	Linux Mint-12, Fedora 16. May work with other distributions but not tested
mac	MacOS Leopard. This version is not available through birusinka site

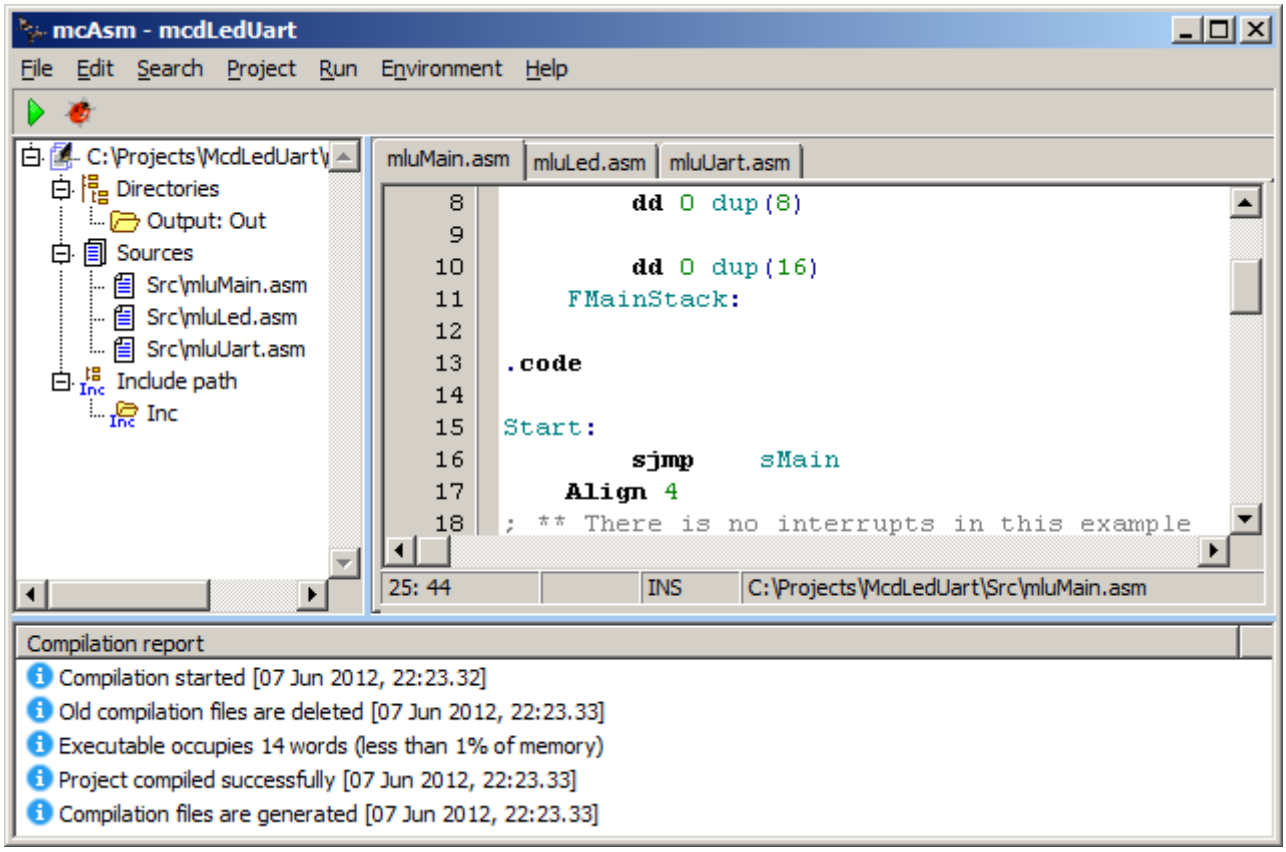
5.2 Installation

There is no installation/deinstallation required. IDE does not modify any system parameters.
Copy executable file 'mcdasm' somewhere to your computer.

5.3 Deinstallation

Delete executable file 'mcdasm' and file 'mcdConfig.cfg' from your computer. File 'mcdConfig.cfg' is created automatically by IDE. This is a text file where IDE saves some settings.

5.4 IDE main window



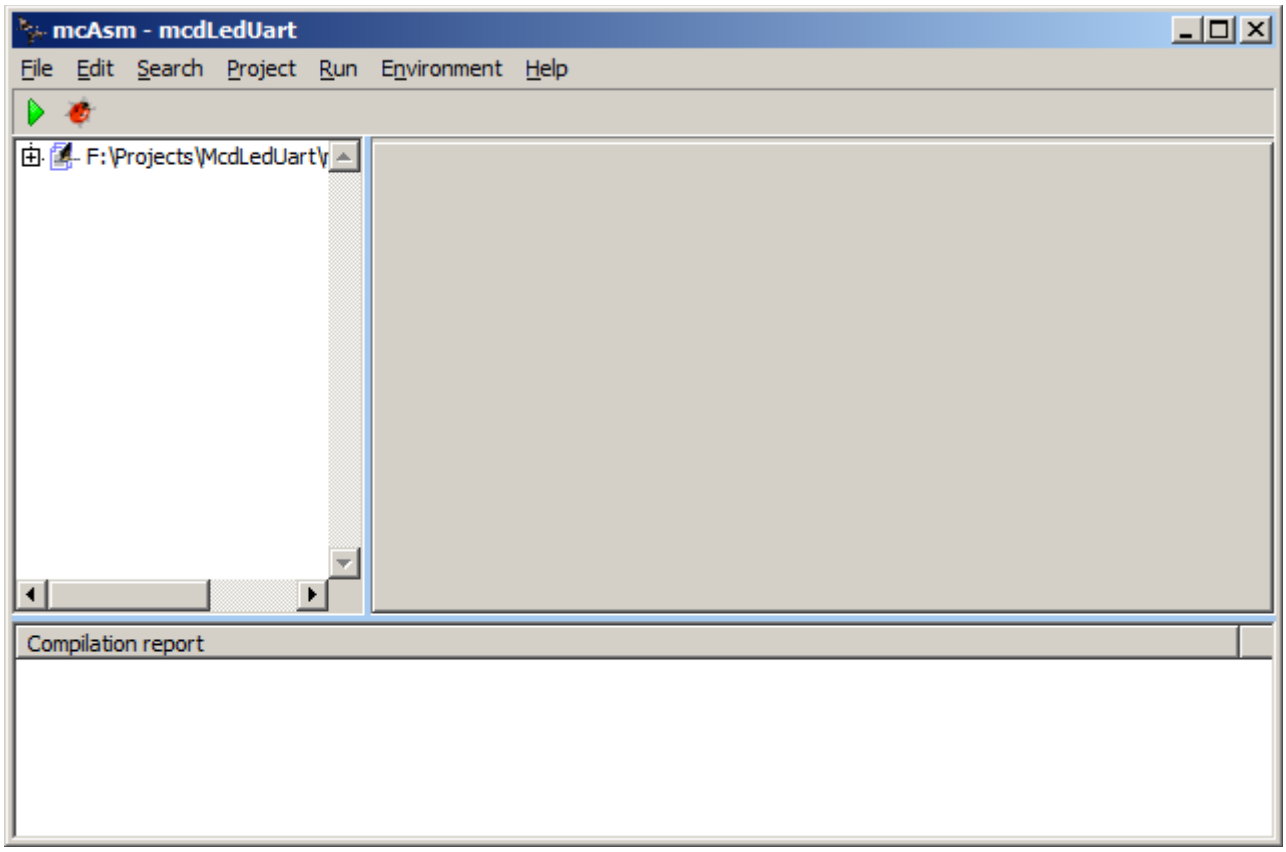
- Top part of the window is a main menu.
- Left part of the window lists all project source files, output directory and include path.
- Right part of the window contains all open files, which can be edited.
- Bottom part shows the compilation result.

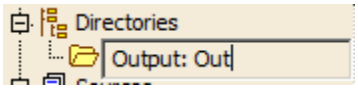
All elements of project have associated pop-up menus. Right click on the element to open it. Double click on the file name opens associated source file.

5.5 Starting a new project

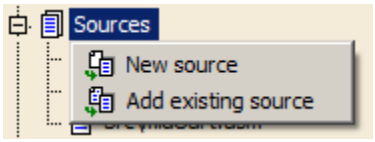
Select **File** → **New** in the menu. A window with an object list will open. Select 'B3 project' and press OK button. You will be requested to choose a directory and give a name for your new project. Select the directory

and type the name of the project. When you press OK button, project will be created. IDE window will look like following.

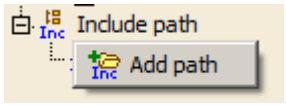




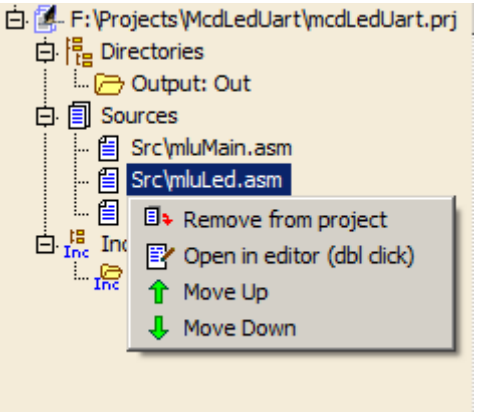
By default, output directory is the same as project directory. You can change it by right clicking 'Output:' element in the left part of the window and change it.



By right clicking 'Sources' element in the left part of the window, append existing source files or specify files which have to be created. The order of source files is important. See chapter 'Memory organization and execution start point' for details.



If there are references to include files in the project, the location of include files must be specified. Right click on 'Include path' element and add locations.

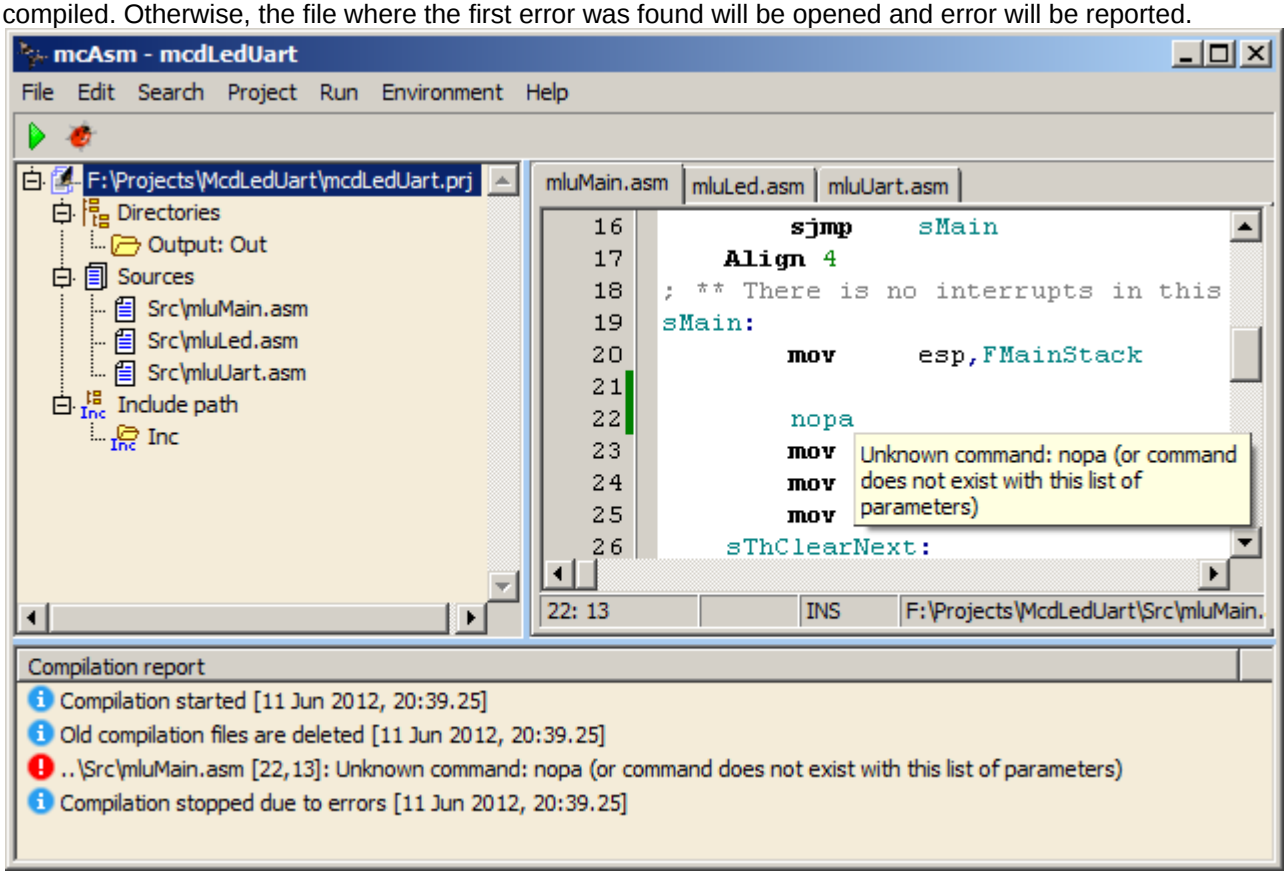


In order to remove source file from the project source file list, right click on the file name. Select 'Remove from project' in the menu. File will only be removed from project, it will not be physically deleted from the disk.

In order to change a position of the file in the list, select 'Move Up' or 'Move Down' in the menu.

5.6 Compiling the project

Press F9 or select **Project** → **Build** in the menu to compile the project. If there is no errors, project will be compiled. Otherwise, the file where the first error was found will be opened and error will be reported.



6 Output files format

Compiler produces different types of output files. All of them have the same name as the project file but differ by the extension. They are generated into the output directory.

Example

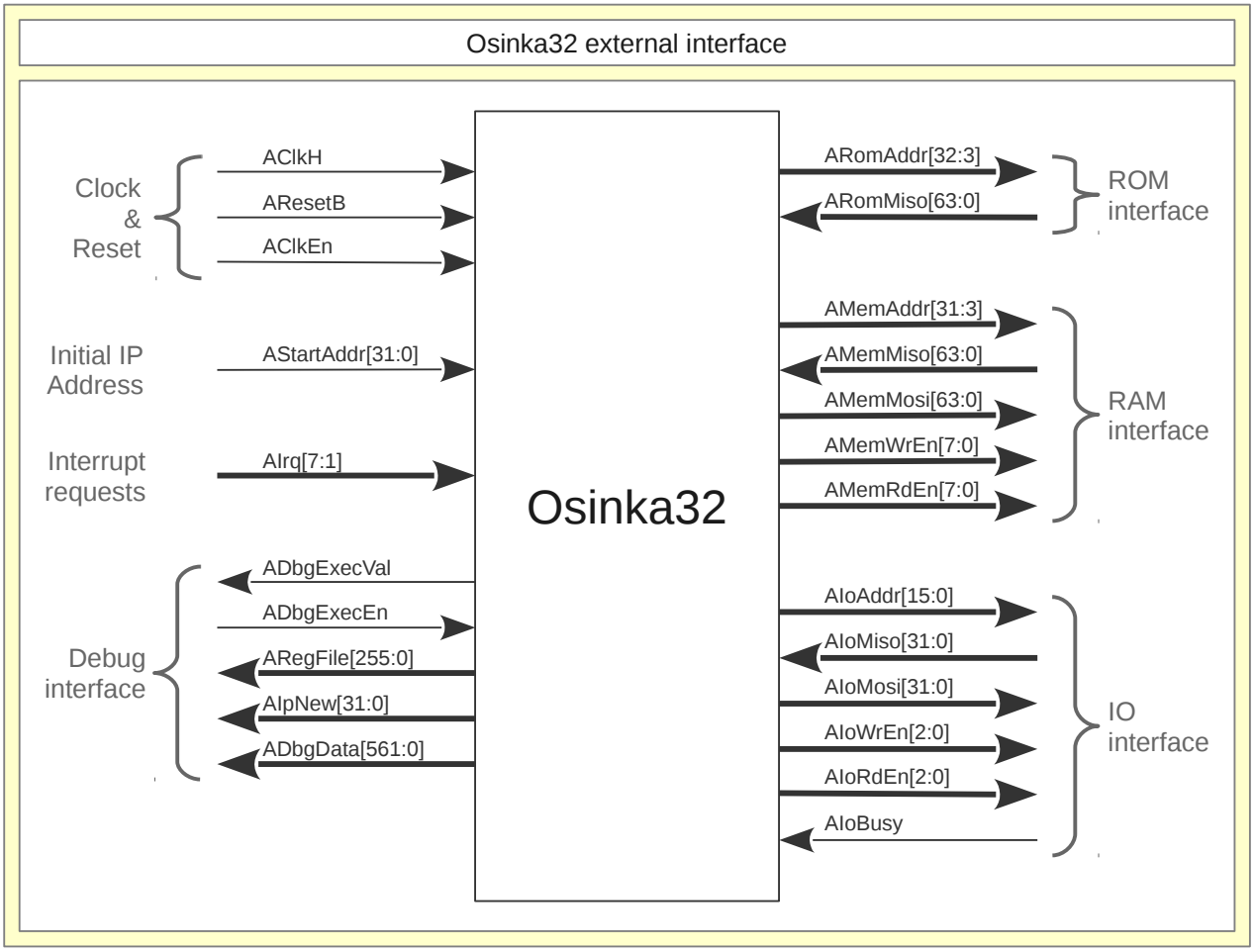
Project name is 'mcdLedUart.prj'. It is located in the directory 'F:\Projects\McdLedUart'. Output directory is 'F:\Projects\McdLedUart\Out'. See picture above. When project is successfully compiled, 5 files will be generated into the output directory:

- mcdLedUart.bin
- mcdLedUart.hex
- mcdLedUart.lst
- mcdLedUart.mif
- mcdLedUart.s90

Output files	
Extension	Purpose
bin	Raw data in binary format. Used for direct memory programming. Unused memory is filled with zeroes
s90	Motorola s90 format. Only executable code is present
hex	Intel-HEX format. Executable code + padding is present
mif	Altera's mif file format. Used to implement RAM-based ROM
lst	Readable listing file. Contains command codes together with source code. Physical address is provided for each line.

7 Hardware implementation

7.1 External interface



Interface signal description			
Signal	Direction	Purpse	Comments
Clock and Reset			
AClkH	input	Main clock signal	
AResetB	input	Global reset signal (active low)	
AClkEn	input	Clock enable signal	
ROM interface			
ARomAddr[32:3]	output	ROM address	Address bus has no lines [2:0]. 64 bit data is read at once
ARomMiso[63:0]	input	ROM data Memory → MCU	
Memory interface			
AMemAddr[31:3]	output	Memory address	Address bus has no lines [2:0], signals memory write enable and memory read enable indicate which part of data is active. Not all bits on data bus can be valid. See memory interface description for details
AMemMosi[63:0]	output	Data MCU → Memory	
AMemMiso[63:0]	input	Data Memory → MCU	
AMemWrEn[3:0]	output	Memory write enable signal	
AMemRdEn[3:0]	output	Memory read enable signal	
I/O interface			
AIoAddr[15:0]	output	IO address	IO address space and memory address space do not overlap. IO bus can address only 65536 addresses. This bus is not mandatory. If no IOs are used, all unused inputs of this bus must be connected to Zero
AIoMosi[31:0]	output	Data MCU → IO	
AIoMiso[31:0]	input	Data IO → MCU	
AIoWrEn[2:0]	output	IO write enable signal	
AIoRdEn[2:0]	output	IO read enable signal	
IRQ			
AIrq[7:0]	input	Interrupt request	

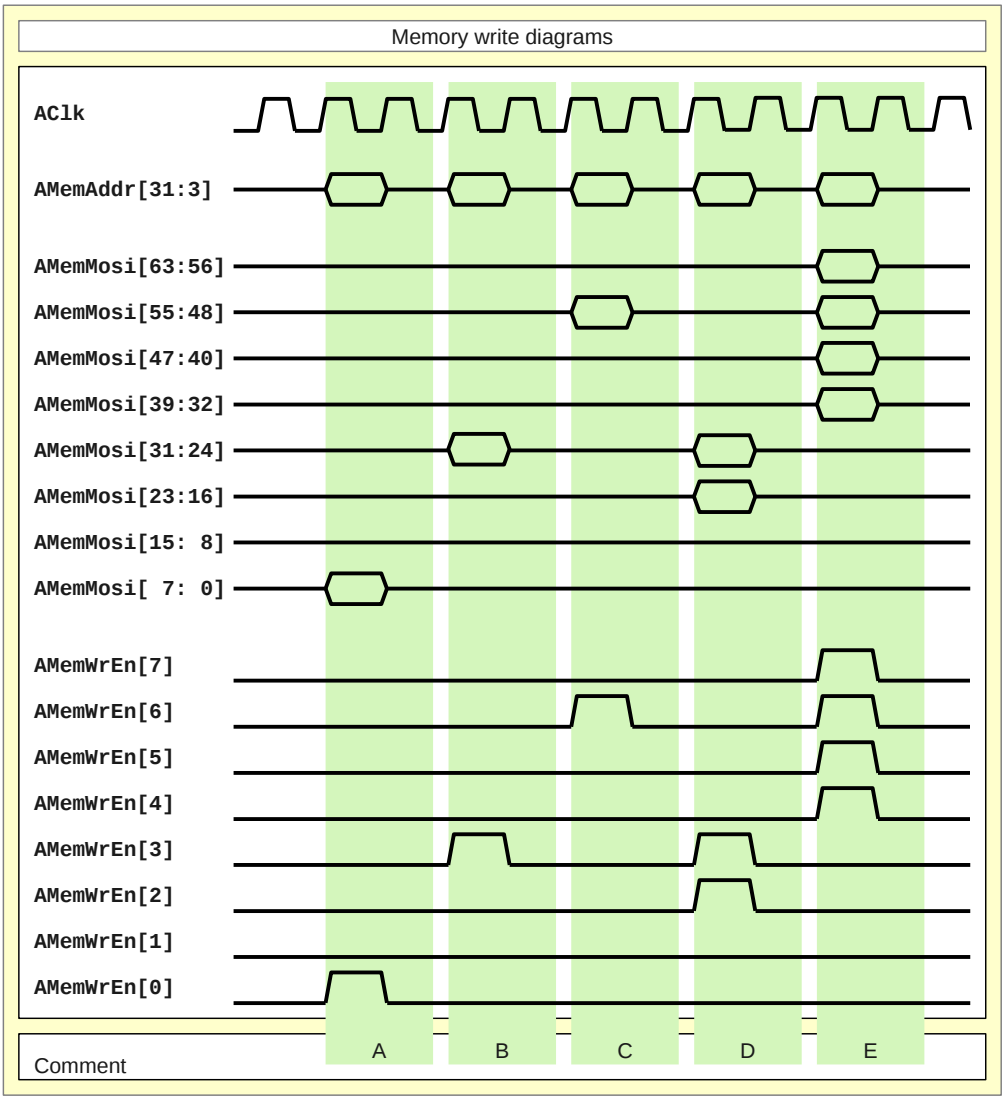
7.2 Clock and Reset

AClkH is the Main clock signal. AClkEn is the clock enable signal delivered to all the Flip-Flops of the design. This signal must be synchronized with AClkH. AResetB is an asynchronous reset signal delivered to all asynchronous reset inputs of all the Flip-Flops of the design.

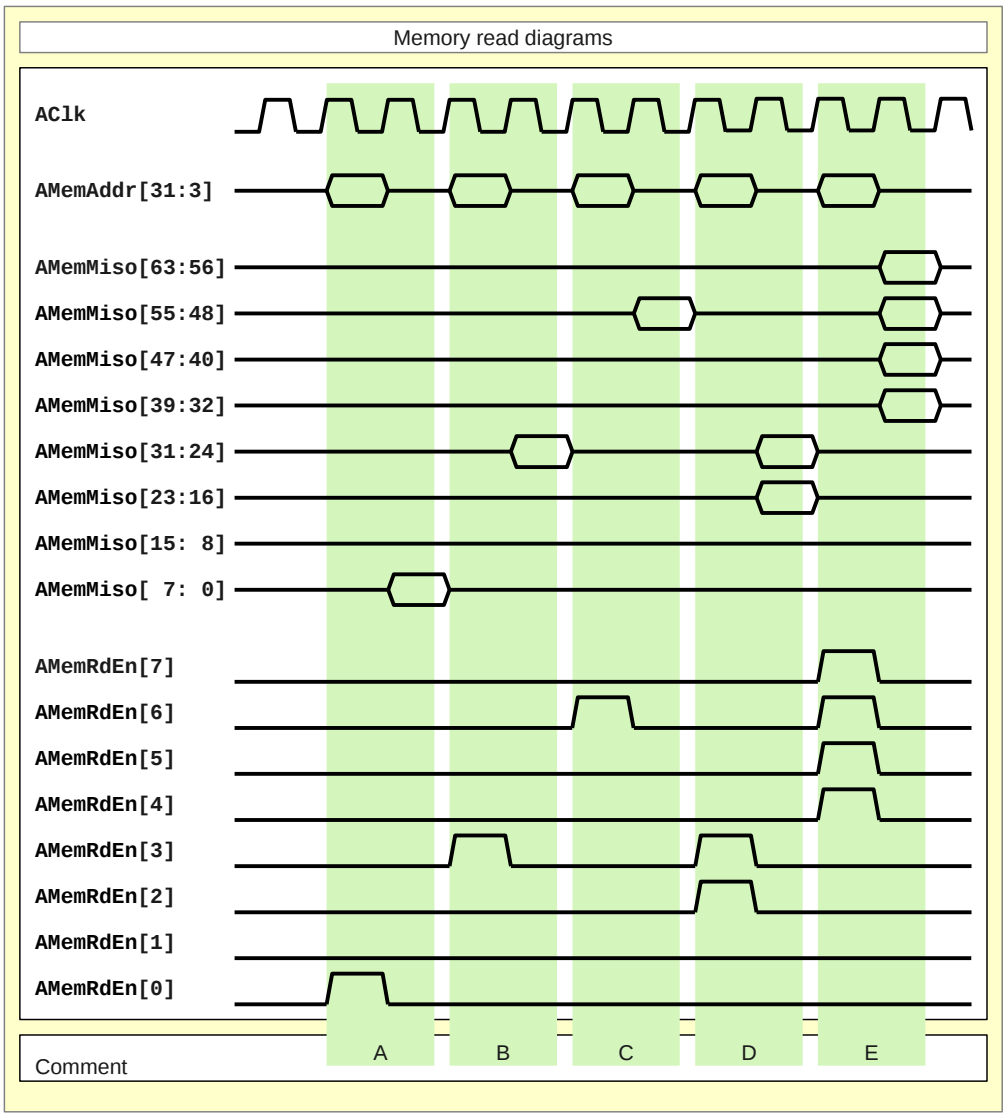
7.3 Memory interface

Memory address bus has no lines [2:0]. Signals Memory Write Enable and Memory Read Enable indicate which part of data is active.

Following pictures illustrate memory write and memory read procedures. See comments for each picture.



Comments for memory write diagrams (see picture above)	
Comment	Explanation
A	Writing of 1 byte. Address[2:0] = 0x00. Only AMemWrEn[0] is active. Data is present in AMemMosi[7:0]
B	Writing of 1 byte. Address[2:0] = 0x03. Only AMemWrEn[3] is active. Data is present in AMemMosi[31:24]
C	Writing of 1 byte. Address[2:0] = 0x06. Only AMemWrEn[6] is active. Data is present in AMemMosi[55:48]
D	Writing of 2 bytes. Address[2:0] = 0x02. AMemWrEn[3:2] are active. Data is present in AMemMosi[31:16]
E	Writing of 4 bytes. Address[2:0] = 0x4. AMemWrEn[7:4] are active. Data is present in AMemMosi[63:32]



Comments for memory read diagrams (see picture above)	
Comment	Explanation
A	Reading of 1 byte. Address[2:0] = 0x00. Only AMemRdEn[0] is active. Data is read from AMemMiso[7:0]
B	Reading of 1 byte. Address[2:0] = 0x03. Only AMemRdEn[3] is active. Data is read from AMemMiso[31:24]
C	Reading of 1 byte. Address[2:0] = 0x06. Only AMemRdEn[6] is active. Data is read from AMemMiso[55:48]
D	Reading of 2 bytes. Address[2:0] = 0x02. AMemRdEn[3:2] are active. Data is read from AMemMiso[31:16]
E	Reading of 4 bytes. Address[2:0] = 0x04. AMemRdEn[7:4] are active. Data is read from AMemMiso[63:32]

Memory interface is optimized for working with synchronous memory blocks. Multi-byte variables must be properly aligned. See alignment chapter for details.

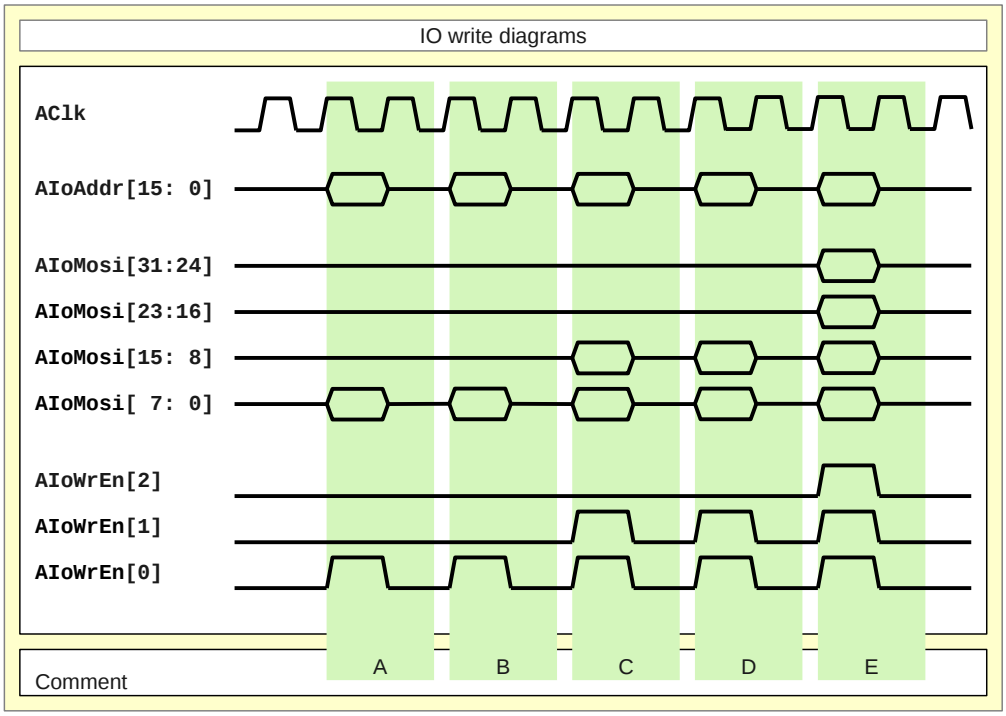
7.4 IO Interface

While memory operations require 2- and 4-byte data to be aligned, for IO operations this is not necessary. Hardware periphery units are responsible for interpreting addressing. 4-, 2- and 1-byte access of the same address can be interpreted differently.

AIoWrEn can only have these values:

- 0 – no operation
- 1 – byte write
- 3 – word write
- 7 – double word (32-bit) write

All other values are RFU.

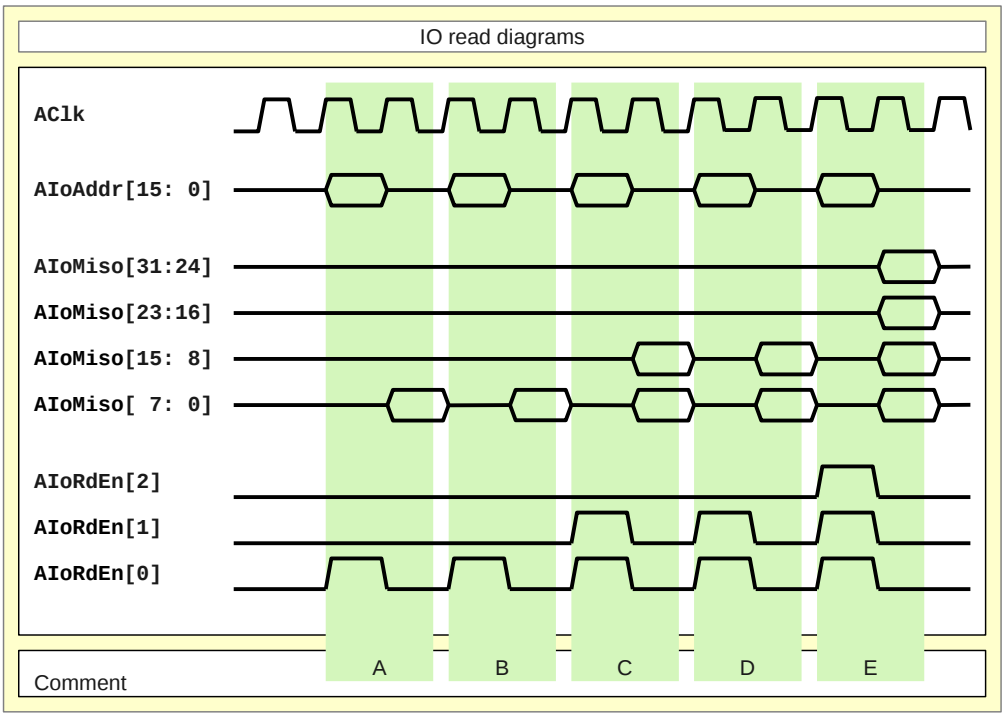


Comments for IO write diagrams (see picture above)	
Comment	Explanation
A	Writing of 1 byte. Address[1:0] = 0x00. AIoWrEn[0] is active. Data is present in AIoMosi[7:0]
B	Writing of 1 byte. Address[1:0] = 0x01. AIoWrEn[0] is active. Data is present in AIoMosi[7:0]
C	Writing of 2 bytes. Address[1:0] = 0x00. AIoWrEn[1:0] is active. Data is present in AIoMosi[15:0]
D	Writing of 2 bytes. Address[1:0] = 0x01. AIoWrEn[1:0] is active. Data is present in AIoMosi[15:0]
E	Writing of 4 bytes. Address[1:0] = 0x00. AIoWrEn[2:0] is active. Data is present in AIoMosi[31:0]

AIoRdEn can only have these values:

- 0 – no operation
- 1 – byte read
- 3 – word read
- 7 – double word (32-bit) read

All other values are RFU.



Comments for IO read diagrams (see picture above)	
Comment	Explanation
A	Reading of 1 byte. Address[1:0] = 0x00. AIoRdEn[0] is active. Data is read from AIoMiso[7:0]
B	Reading of 1 byte. Address[1:0] = 0x01. AIoRdEn[0] is active. Data is read from AIoMiso[7:0]
C	Reading of 2 bytes. Address[1:0] = 0x00. AIoRdEn[1:0] is active. Data is read from AIoMiso[15:0]
D	Reading of 2 bytes. Address[1:0] = 0x01. AIoRdEn[1:0] is active. Data is read from AIoMiso[15:0]
E	Reading of 4 bytes. Address[1:0] = 0x00. AIoRdEn[2:0] is active. Data is read from AIoMiso[31:0]

Disclaimer

The information contained in this document is for general information purposes only. The information is provided by birusinka.com and while we try to keep the information up to date and correct, we make no representations or warranties of any kind, express or implied, about the completeness, accuracy, reliability, suitability or availability with respect to the document or the information, products, services, or related content for any purpose. Any reliance you place on such information is therefore strictly at your own risk.

In no event will we be liable for any loss or damage including without limitation, indirect or consequential loss or damage, or any loss or damage whatsoever arising from loss of data or profits arising out of, or in connection with, the use of this document.